Report: Agent Selection Algorithm

Introduction

Agent selection algorithm aims to match user query with ‘best’ agent.

Goal:

- Input: (query, set of agent descriptions), and (response time, input cost, output cost, average rating, rated responses, popularity)
- Output: 'Best' agent

Feature engineering:

- **Quality score:** average rating provide insight to how users feel on ave, it is unreliable when rated_responses is low. So we scale average_rating with rated_responses and call it quality_score:

$$\text{Quality Score} = \frac{\text{average\_rating} \times \text{rated\_responses}}{\text{rated\_responses} + k}$$

  where smaller k values make the quality score converge to the average rating faster with respect to rated responses. When k = 0, the quality score equals the average rating.

- **Adjusted quality score:** Quality score is still flawed. If there agent with 0 rated_reponse, quality_score become zero giving agent not chance to be selected. We would want those agent some chance and come up with adjusted_quality_score:

$$adjusted\_quality\_score = \frac{\text{average\_rating} \times \text{rated\_responses} + \text{baseline\_rating} \times k}{\text{rated\_responses} + k}$$

  This still converge to average_rating and give newer agents a chance. However, this metric dilute the impact of well-rated agent. If we set the `baseline_rating` to 5, any agent with zero rated responses would receive a `quality_score` of 5, making them indistinguishable from agents who consistently earn 5-star ratings from users. So lower `baseline_rating` is recommended.

- **log popularity:** The impact of popularity should be more significant for a change from 0 to 100 users than from 10,000 to 10,100 users. So we scale them by log:

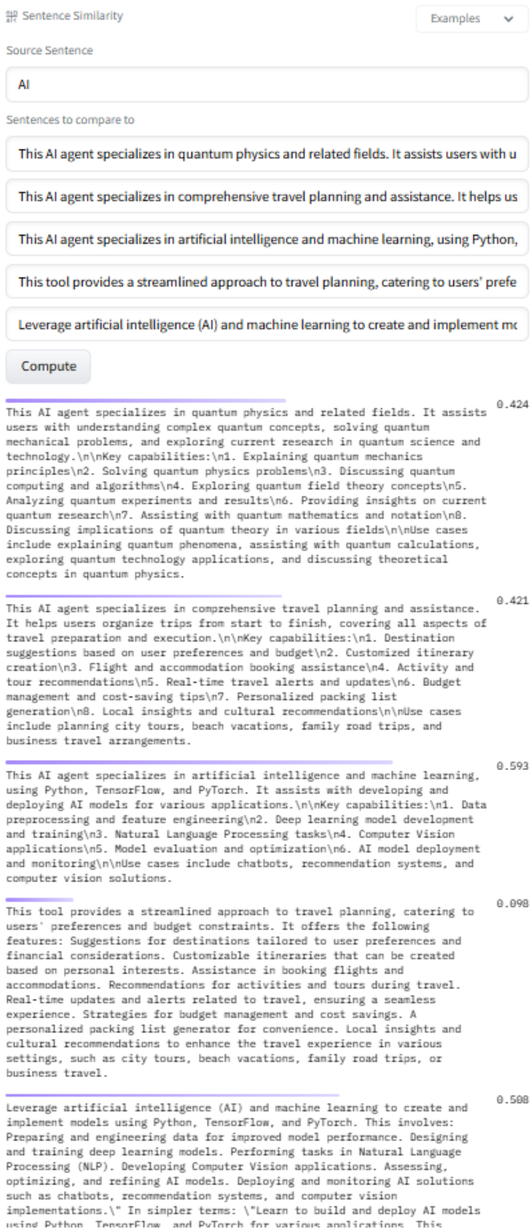
$$log\_popularity\_score = log(popularity + 1)$$

- **total estimated cost:** Ideally, we'd use average input and output tokens to scale the total estimated cost, but we currently don’t have this data.

$$Total\_estimated\_cost = input\_cost + output\_cost.$$

- **Processed description:** Instead of using the given description directly we
  1. remove some phrase that may cause bias getting cleaned_description
  2. then rephrase the cleaned_description using LLM.

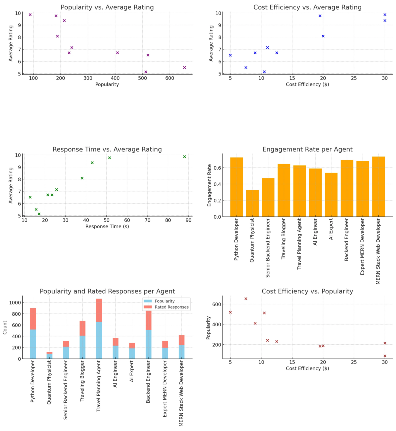
Why? Without processing, phrases like “This AI agent” may cause irrelevant matches. For example below, a user query “AI” could match both a quantum physicist and a travel agent simply due to the generic “AI” mention. Processing eliminates these false positives, ensuring more accurate matching.



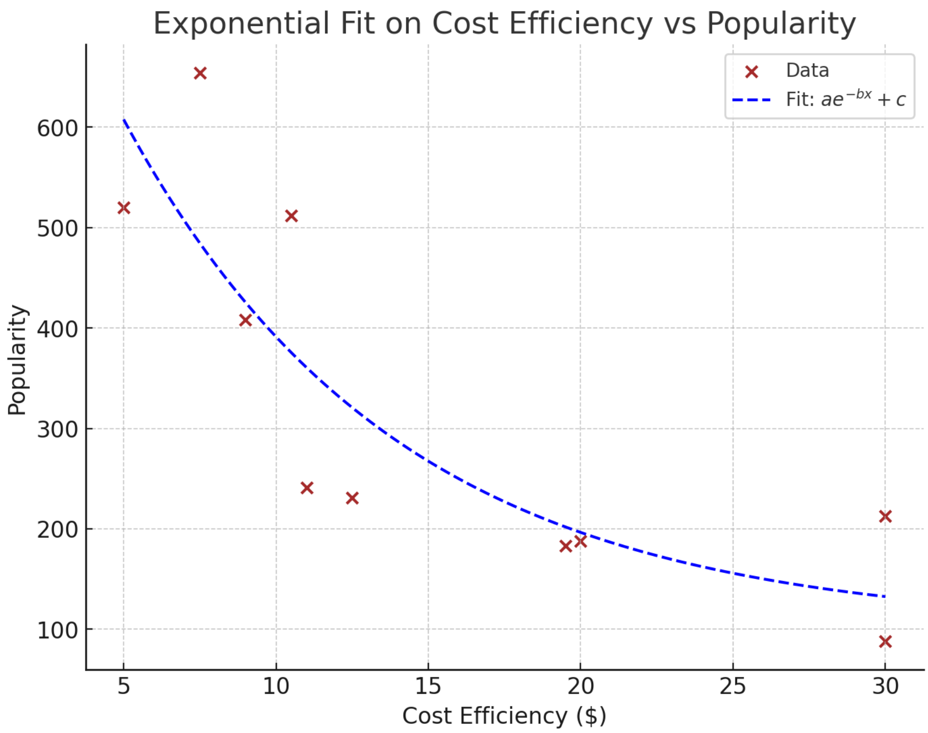
Feature analysis (on non-benchmark data)

This is done on data in `data/agents` folder

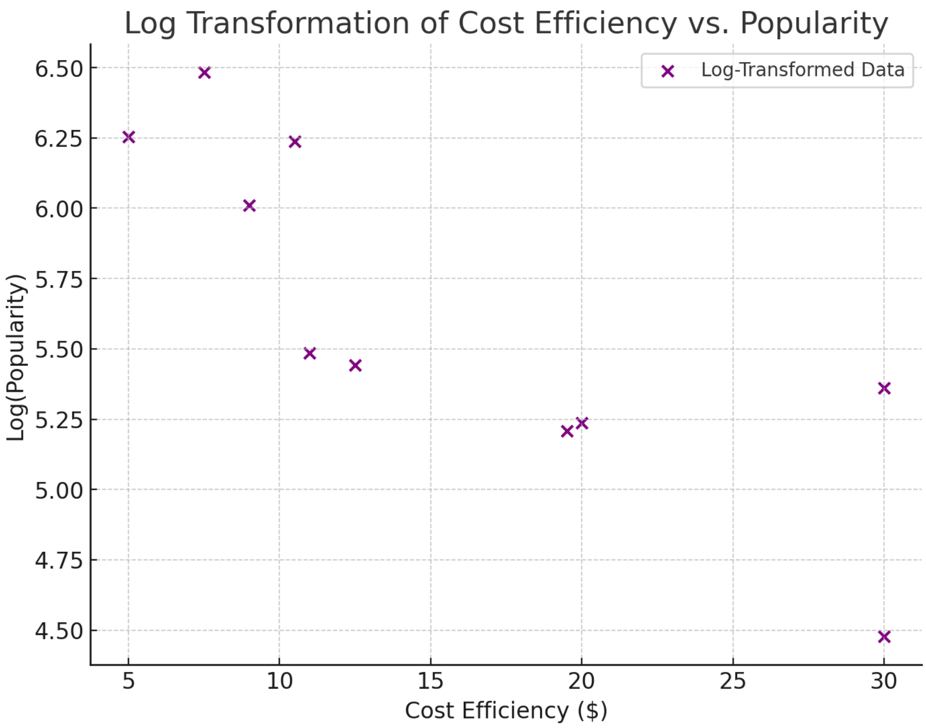
There seem to be linear relationship between many of these features. We are not sure how we should utilize them though.(cost efficiency is total estimated cost.... I know it was a bad naming)



1. Agent with low average rating seem to have high popularity. Very counter intuitive.
2. The more expensive agent seem to have higher average rating. make sense.
3. Agent with higher response time seem to have high average rating.
4. From 2 and 3, There must be relationship between total estimated cost and repsonse time.

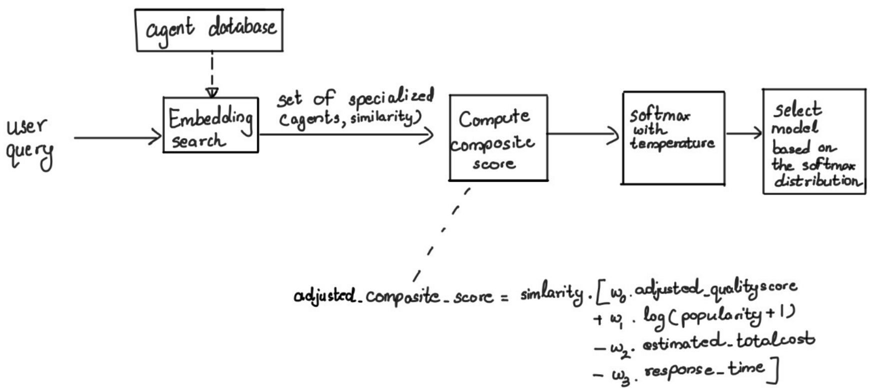


- Users seem to query agent with lower cost more, exponentially. This support our decision to use log scale on popularity.



- log_popularity seem to result in linear relationship.

Agent selection



Algorithm goes like this:

1. Perform embedding search. May be single stage or two-stage.
2. (Optional) Filter agents based on similarity score.
3. Compute adjusted composite score of top_n agents.
4. Apply softmax function on composite score and sample from the (uncalibrated) probability distribution.

- Note: In benchmark, we just select agent based on similarity score because sampling add randomness.

After discussing features, we should look into our main algorithm.

- Embedding functions
    - **Embedding function:** We use sentence-transformer text embedding (all-mpnet-base-v2) with TFIDF.
        - Neural network for *semantic search*. TFIDF for *lexical search* (with snowflake stem preprocessing). lexical search is particularly suitable for this task, since user query is (probably) not very long.
        - We also try SPLADE which give good result: SPLADE for Sparse Vector Search Explained | Pinecone
        - agent descriptions are precomputed.

- **Similarity function:** We use cosine similarity (why? Pretrained Models — Sentence Transformers documentation)
    - In hybrid search, linear weighting usually does not work well since two scores may be on different scale.

$$\text{score} = \alpha \times score_0 + (1 - \alpha) \times score_1$$

- RRF: https://medium.com/@devalshah1619/mathematical-intuition-behind-reciprocal-rank-fusion-rrf-explained-in-2-mins-002df0cc5e2a

- **Efficient embedding search**: We use **Hierarchical Navigable Small Worlds (HNSW)** algorithm. Which trade setup time and high memory usage for fast search, add, remove time. We also try some other search algorithm using faiss (Faiss: The Missing Manual | Pinecone). However, we think HNSW is ok for now in term of both efficiency and accuracy.
    - Chromadb use HNSW so we just stick with it.

- Composite score

  While similarity between query and agent is important, but when many agents have similar description, additional metrics can help decide between them.

    - We combine `adjusted_quality_score`, `log_popularity`, `estimated_total_cost`, and `response_time` in a weighted sum to calculate the composite score. These weights are currently intuition-based.
    - Ideally, these weights would be optimized based on user preferences or measurable objectives. If relationships are complex, a machine learning model could replace the composite score.
    - Then, we have 2 possible approaches:
        1. **Fixed Agent Pool:** Use a fixed number of agents and multiply `composite_score` by `similarity_score` to obtain an `adjusted_composite_score`. This approach prioritizes query-description similarity while still factoring in other metrics.
        2. **Similarity-Filtered Pool:** Filter agents based on a threshold relative to the maximum similarity score, ensuring only agents highly similar to the query are considered.

```
max_similarity = max(agent["Similarity Score"] for agent in scored_agents)
filtered_agents = [agent for agent in scored_agents if agent["Similarity Score"]
> 0.8 * max_similarity]
```

Selection:

When multiple agents have exactly same descriptions, one may have higher ratings or popularity than others. We would want to prioritize agents with higher scores, while allowing others a chance as well.

- A straightforward approach is to select the agent with the highest `composite_score`. However, this may exclude agents with similar capabilities but slightly lower ratings or popularity.
- Alternatively, applying a softmax function to the `composite_score` generates a probability distribution, enabling us to sample agents and give those with slightly lower scores a chance of selection.

- Why not cross-encoder?
  - We tried but results are not as good as embedding in our case. It does improve some cases though as we will see in results.
  - Most cross-encoders are optimized for sentence pairs with exact meaning matches, which is different from our goal of matching user questions with agent descriptions.
    - For reference, many cross-encoder models are trained on datasets like SNLI and MultiNLI, where sentence similarity is key.

Result

In the benchmark, we selected the agent with the highest similarity score. For models using processed descriptions, we ran 5 trials to account for output variability introduced by the LLM.

We used the following models:

- mpet: all-mpnet-base-v2
- allminiLM: all-MiniLM-L6-v2
- **Embedding Model 2**: Ensemble of Transformer + SPLADE
- **Cross-Encoder Model**: ms-marco-MiniLM-L-6-v2

Description	Processed Description	Run 1	Run 2	Run 3	Run 4	Run 5	Average
mpnet	0	0.458333	0.458333	0.458333	0.458333	0.458333	0.45833
allminiLM	0	0.541667	0.541667	0.541667	0.541667	0.541667	0.54166
tfidf	0	0.125000	0.125000	0.125000	0.125000	0.125000	0.12500
SPLADE	0	0.416667	0.416667	0.416667	0.416667	0.416667	0.41666
tfidf + mpet	0	0.500000	0.500000	0.500000	0.500000	0.500000	0.50000
tfidf + allminiLM	0	0.541667	0.541667	0.541667	0.541667	0.541667	0.54166
tfidf+SPLADE	0	0.458333	0.458333	0.458333	0.458333	0.458333	0.45833
SPLADE + mpet	0	0.500000	0.500000	0.500000	0.500000	0.500000	0.50000
allminiLM+mpet	0	0.500000	0.500000	0.500000	0.500000	0.500000	0.50000
Cross Encoder	0	0.5833	0.5833	0.5833	0.5833	0.5833	0.5833
SPLADE + allminiLM	0	0.500000	0.500000	0.500000	0.500000	0.500000	0.50000
mpnet	1	0.625000	0.583333	0.666667	0.625000	0.625000	0.62500
allminiLM	1	0.375000	0.583333	0.666667	0.666667	0.583333	0.57500
tfidf	1	0.166667	0.166667	0.166667	0.208333	0.166667	0.17500
SPLADE	1	0.541667	0.583333	0.625000	0.708333	0.625000	0.61666
tfidf + mpet	1	0.583333	0.583333	0.625000	0.625000	0.666667	0.61666
tfidf + allminiLM	1	0.333333	0.583333	0.625000	0.666667	0.583333	0.55833
tfidf+SPLADE	1	0.500000	0.583333	0.583333	0.708333	0.666667	0.60833
SPLADE + mpet	1	0.583333	0.625000	0.750000	0.750000	0.750000	0.69166

SPLADE + allminiLM	1	0.541667	0.583333	0.708333	0.666667	0.708333	0.64166
allminiLM+mpet	1	0.666667	0.625000	0.708333	0.666667	0.708333	0.67500
Cross Encoder	1	0.5833	0.5833	0.5417	0.5833	0.6250	0.58333

Analysis

- **Effect of Processed Descriptions**:: Processing descriptions significantly improves the embedding model's performance.
- mpet and SPLADE work great with processed description.
- SPLADE + mpet get highest average accuracy of 0.692 and peak at 0.75.

Accuracy

We analyze some mistakes that SPADE + mpet makes:

Query	Predicted Agent	Expected Agent	**Comment**
Setup authentication system with JWT	Web Architect	Python Backend Developer	This prediction make sense. JSON Web Tokens (JWT) are a powerful and flexible tool for secure ***authentication*** and data exchange between parties.
How to handle exceptions and system calls in Python?	Python Systems Architect	Python Developer	I agree that Python Developer would be better suit here. However, both are fine.
Design ECS architecture for large worlds	AI Architect	Engine Developer	ECS architecture is quite new. It may not be in the training set. This may get match just because of the keyword `architecture`
Explain transformer architecture basics	Web Architect	AI Researcher	
Implement the latest Vision Transformer architecture	Deep Learning Engineer	Machine Learning Researcher	I think Deep Learning Engineer is actually the better answer here?
Derive theoretical bounds for AI capability limits	AI Architect	AGI Researcher	I think both seem almost equally fine.

These results show that even though our accuracy may not be very high (~0.69), there are cases where the predicted answer is contextually reasonable, even if it doesn't match the exact expected agent.